



Università degli Studi di Bologna
Corso di Laurea in Ingegneria Informatica

Garbage Collection

Ingegneria del Software T

Prof. MARCO PATELLA

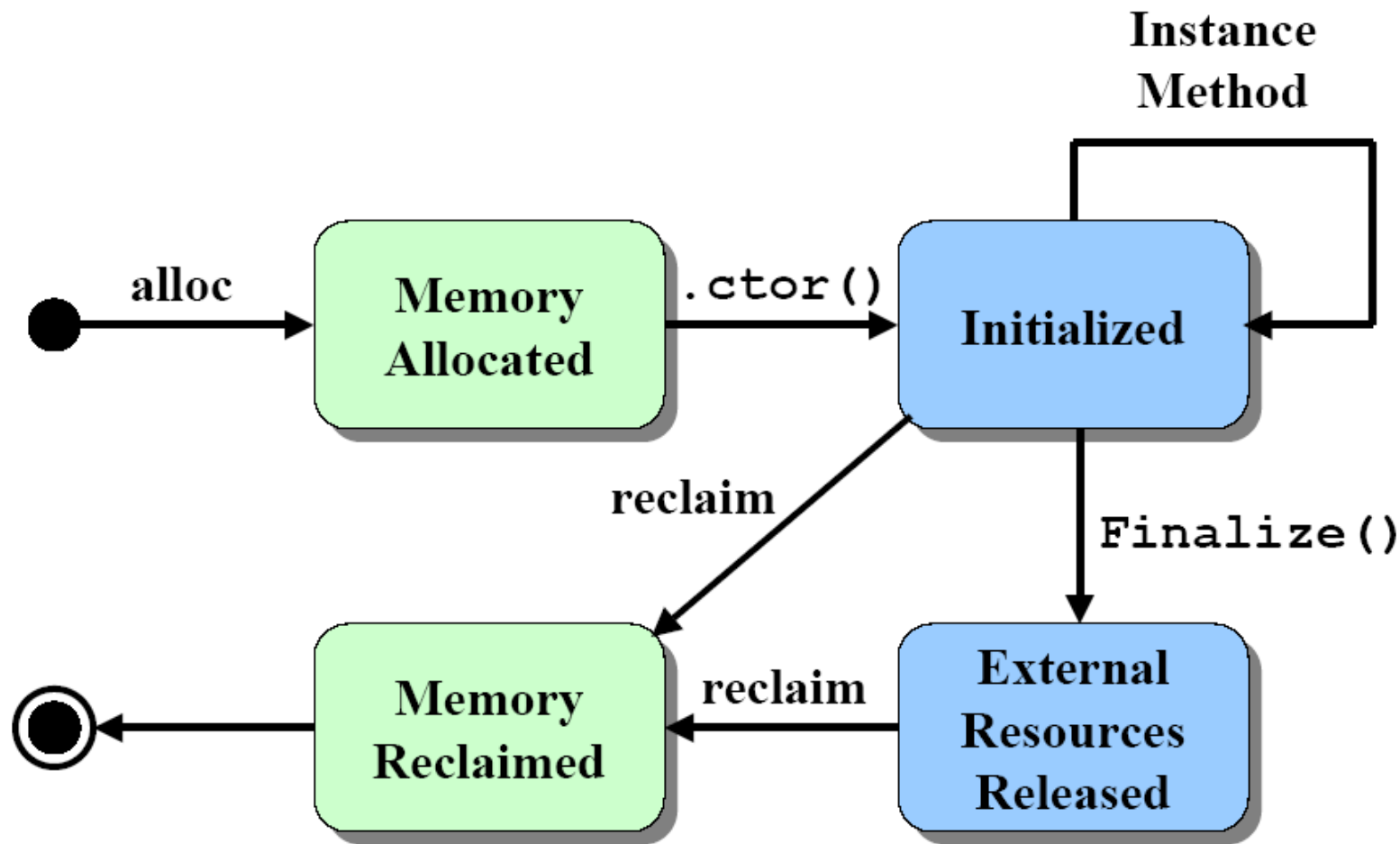
Dipartimento di Informatica – Scienza e Ingegneria (DISI)

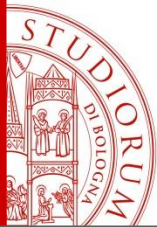


Utilizzo di un oggetto

- In un ambiente *object-oriented*, ogni oggetto che deve essere utilizzato dal programma
 - È descritto da un tipo
 - Ha bisogno di un'area di memoria dove memorizzare il suo stato
- Passi per utilizzare un oggetto di tipo riferimento:
 - **Allocare memoria** per l'oggetto
 - **Inizializzare la memoria** per rendere utilizzabile l'oggetto
 - **Usare l'oggetto**
 - **Eseguire un clean up** dello stato dell'oggetto, se necessario
 - **Liberare la memoria**

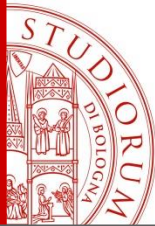
Ciclo di vita di un oggetto





Allocazione della memoria

- In C:
 - `malloc` (`calloc`, `realloc`)
- In C++:
 - `malloc` (`calloc`, `realloc`)
 - `new`
- In Java:
 - `new`
- In IL:
 - `newobj`
- In C#:
 - `newobj`



Inizializzazione della memoria

- Definite Assignment: a ogni variabile deve essere sempre assegnato un valore prima che essa venga utilizzata
 - il compilatore deve assicurarsi che ciò sia sempre verificato
 - Data-flow analysis del codice
- valori di default
 - usati in generale per tipi valore
 - ad esempio, in Java le variabili di classe, locali e i componenti di un array sono inizializzati al valore di default, **non** le variabili di istanza (perché?)
- costruttore
 - usato per i tipi classe (Java, C++, C#)



Definite Assignment

```
int k;  
if (v > 0 && (k = System.in.read()) >= 0)  
    System.out.println(k);
```

Corretto?

```
int k;  
while (n < 4) {  
    k = n;  
    if (k >= 5) break;  
    n = 6;  
}  
System.out.println(k);
```

Corretto?



Definite Assignment

```
int k;  
while (true) {  
    k = n;  
    if (k >= 5) break;  
    n = 6;  
}  
System.out.println(k);
```

Corretto?

```
int k;  
int n=5;  
if (n>2) k=3;  
System.out.println(k);
```

Corretto?



Clean up dello stato

- In C++/C#:
 - distruttore (più propriamente, finalizzatore): ~{nome della classe}
 - unico, non ereditabile, no overload, senza parametri e modificatori
 - invocato automaticamente alla distruzione dell'oggetto (non può essere invocato)
- In java:
 - **finalize()**
 - metodo di **Object**
 - invocato automaticamente alla distruzione dell'oggetto (non può essere invocato)
 - il momento in cui viene invocato un finalizzatore dipende dalla JVM



Liberazione della memoria

- In C:
 - `free()`
- In C++:
 - `free()`
 - `delete`
- In java/C#: garbage collector (GC)



Garbage Collection

- Modalità automatica di rilascio delle risorse utilizzate da un oggetto
- Migliora la stabilità dei programmi
 - Evita errori connessi alla necessità, da parte del programmatore, di manipolare direttamente i puntatori alle aree di memoria
- Pro:
 - dangling pointer
 - doppia de-allocazione
 - memory leak
- Contro:
 - aumentata richiesta risorse di calcolo
 - incertezza del momento in cui viene effettuata la GC
 - rilascio della memoria non deterministico



Garbage Collection

- Strategie disponibili:
 - Tracing
 - determinare quali oggetti sono (potenzialmente) *raggiungibili*
 - eliminare gli oggetti non raggiungibili
 - Reference counting
 - ogni oggetto contiene un contatore che indica il numero di riferimenti a esso
 - la memoria può essere liberata quando il contatore raggiunge lo 0
 - Escape analysis
 - si spostano oggetti dallo heap allo stack
 - l'analisi viene effettuata a compile-time in modo da stabilire se un oggetto, allocato all'interno di una subroutine, non è accessibile al di fuori di essa
 - riduce il lavoro del GC



GC: Reference counting

- Svantaggi:
 - Cicli di riferimenti
 - se due oggetti si referenziano a vicenda, il loro contatore non raggiungerà mai 0
 - Aumento dell'occupazione di memoria
 - Riduzione della velocità delle operazioni sui riferimenti
 - ogni operazione su un riferimento deve anche incrementare/decrementare i contatori
 - Atomicità dell'operazione
 - ogni modifica a un contatore deve essere resa operazione atomica in ambienti multi-threaded
 - Assenza di comportamento real-time
 - ogni operazione sui riferimenti può (potenzialmente) causare la de-allocazione di diversi oggetti
 - il numero di tali oggetti è limitato solamente dalla memoria allocata



GC: Tracing

- Siano **p** e **q** due oggetti
- Sia **q** un oggetto raggiungibile
- Diremo che **p** è raggiungibile in maniera ricorsiva se e solo se:
 - esiste un riferimento a **p** tramite **q**
 - ovvero **p** è raggiungibile attraverso un oggetto, a sua volta raggiungibile
- Un oggetto è pertanto raggiungibile in due soli casi:
 - è un oggetto **radice**
 - creato all'avvio del programma (oggetto *globale*)
 - creato da una sub-routine (oggetto *scope*, riferito da variabile sullo stack)
 - è referenziato da un oggetto raggiungibile
 - la raggiungibilità è una *chiusura transitiva*



GC: Tracing

- La definizione di **garbage** tramite la raggiungibilità **non** è ottimale
 - può accadere che un programma utilizzi per l'ultima volta un certo oggetto molto prima che questo diventi irraggiungibile
- Distinzione:
 - garbage **sintattico**
(oggetti che il programma non *può* raggiungere)
 - garbage **semantico**
(oggetti che il programma non *vuole* più usare)
 - problema solo parzialmente decidibile (quindi?)

```
Object x = new Foo();  
Object y = new Bar();  
x = new Quux(); // qui l'oggetto Foo è garbage sintattico  
if(x.check_something())  
    x.do_something(y); // qui y *potrebbe* essere garbage  
semantico
```



Allocazione della memoria

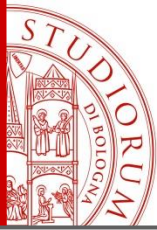
- In fase di inizializzazione di un processo, il CLR
 - Riserva una regione contigua di spazio di indirizzamento *managed heap*
 - Memorizza in un puntatore (**NextObjPtr**) l'indirizzo di partenza della regione



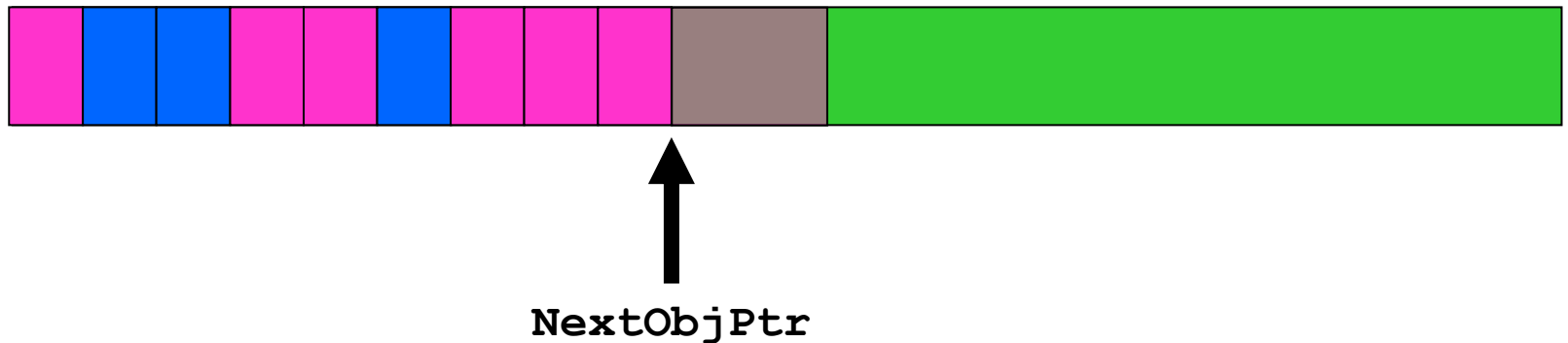


Allocazione della memoria




- Quando deve eseguire una **newobj**, il CLR
 - Calcola la dimensione in *byte* dell'oggetto e aggiunge all'oggetto due campi di 32 (o 64) bit
 - Un puntatore alla tabella dei metodi
 - Un campo **SyncBlockIndex**
 - Controlla che ci sia spazio sufficiente a partire da **NextObjPtr**
 - in caso di spazio insufficiente:
 - *garbage collection*
 - **OutOfMemoryException**
 - **thisObjPtr = NextObjPtr;**
 - **NextObjPtr += sizeof(oggetto);**
 - Invoca il costruttore dell'oggetto (**this** \equiv **thisObjPtr**)
 - Restituisce il riferimento all'oggetto



Allocazione della memoria



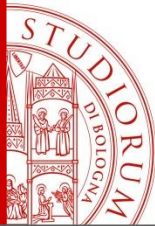
Tecnica di allocazione
completamente diversa
da quella del C/C++

-  Oggetti “vivi”
-  Oggetti non raggiungibili
-  Spazio libero



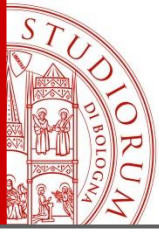
Garbage Collector

- Verifica se nell'*heap* esistono oggetti non più utilizzati dall'applicazione
 - Ogni applicazione ha un insieme di radici (*root*)
 - Ogni radice è un puntatore che contiene l'indirizzo di un oggetto di tipo riferimento oppure vale `null`
 - Le radici sono:
 - Variabili globali e *field* statici di tipo riferimento
 - Variabili locali o argomenti attuali di tipo riferimento sugli *stack* dei vari *thread*
 - Registri della CPU che contengono l'indirizzo di un oggetto di tipo riferimento
 - **Gli oggetti “vivi”** sono quelli **raggiungibili** direttamente o indirettamente dalle radici
 - **Gli oggetti *garbage*** sono quelli **NON raggiungibili** direttamente o indirettamente dalle radici



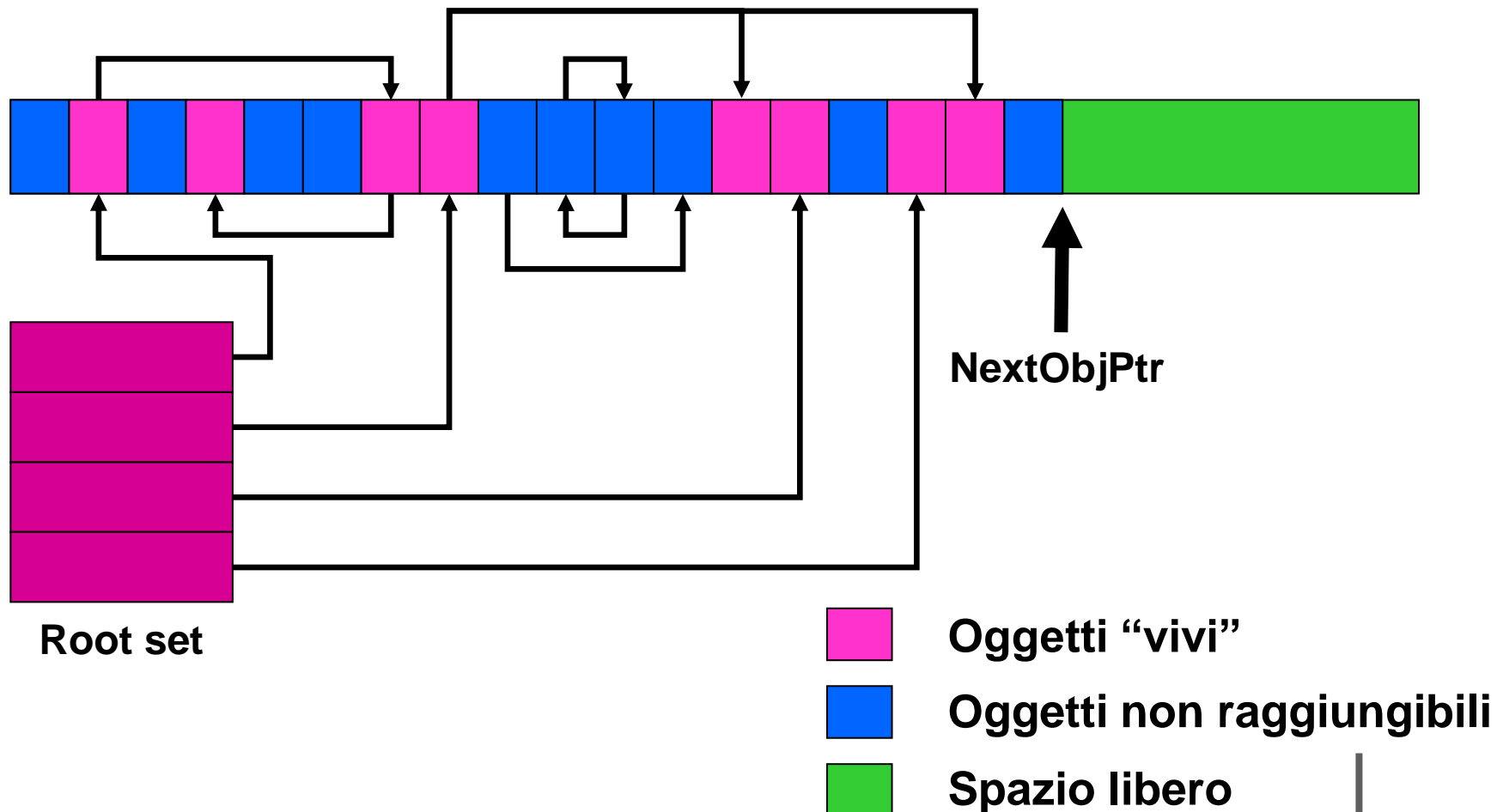
Garbage Collector

- Quando parte, il GC ipotizza che tutti gli oggetti siano *garbage*
- Quindi, scandisce le radici e per ogni radice **marca**
 - l'eventuale oggetto referenziato e
 - Tutti gli oggetti a loro volta raggiungibili a partire da tale oggetto
- Se durante la scansione incontra un oggetto già marcato in precedenza, lo salta
 - sia per motivi di prestazioni
 - sia per gestire correttamente riferimenti ciclici
- Una volta terminata la scansione delle radici, tutti gli oggetti NON marcati sono non raggiungibili e quindi *garbage*



Garbage Collector

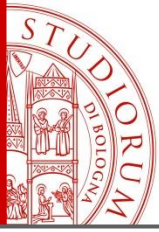
Fase 1: Mark





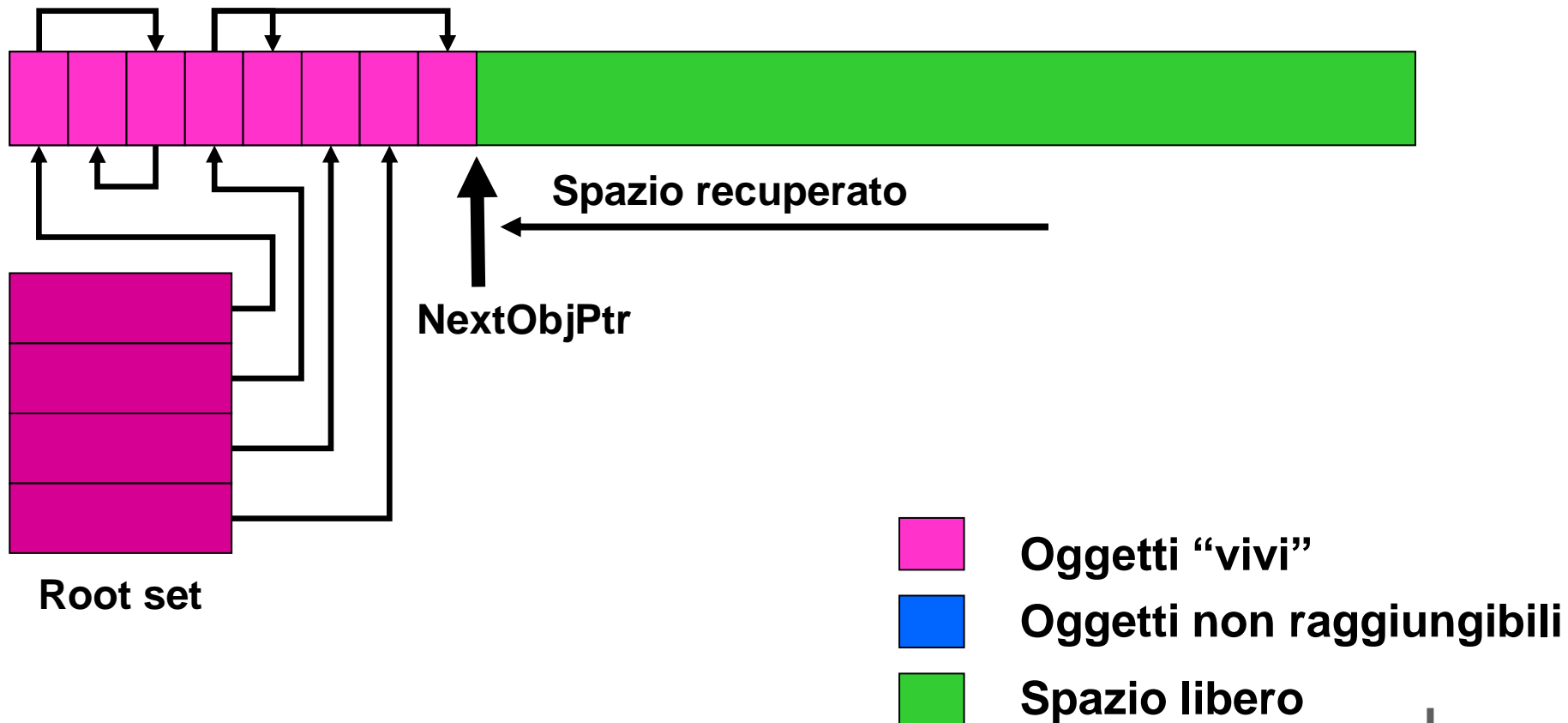
Garbage Collector

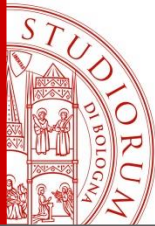
- Rilascia la memoria usata dagli oggetti non raggiungibili
- **Compatta** la memoria ancora in uso, **modificando nello stesso tempo tutti i riferimenti agli oggetti spostati!**
- Unifica la memoria disponibile, aggiornando il valore di **NextObjPtr**
- Tutte le operazioni che il GC effettua sono possibili in quanto
 - Il tipo di un oggetto è sempre noto
 - È possibile utilizzare i metadati per determinare quali *field* dell'oggetto fanno riferimento ad altri oggetti



Garbage Collector

Fase 2: Compact





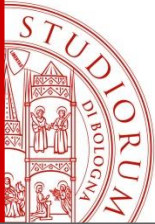
Finalization

- Non è responsabilità del GC, ma del programmatore
- Se un oggetto contiene esclusivamente
 - tipi valore e/o
 - riferimenti a oggetti *managed*

(maggior parte dei casi), non è necessario eseguire alcun codice particolare

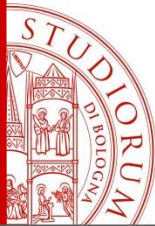
- Se un oggetto contiene almeno un riferimento a un oggetto *unmanaged* (in genere, una risorsa del S.O.)
 - file, connessione a database, socket, mutex, bitmap, ...

è necessario eseguire del codice per rilasciare la risorsa, prima della deallocazione dell'oggetto



Finalization

- Ad esempio, un oggetto di tipo **System.IO.FileStream**
 - Prima deve aprire un file e memorizzare in un suo *field* l'*handle* del file (una risorsa di S.O. *unmanaged*)
 - Quindi usa tale *handle* nei metodi **Read** e **Write**
 - Infine, deve rilasciare l'*handle* nel metodo **Finalize**
- In C#
 - NON è possibile definire il metodo **Finalize**
 - È necessario definire un **distruttore** (sintassi C++)



Finalization

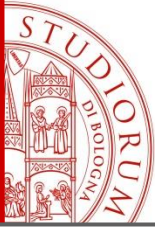
```
public class OSHandle
{
    // Field contenente l'handle della risorsa unmanaged
    private readonly IntPtr _handle;

    public IntPtr Handle
    { get { return _handle; } }

    public OSHandle(IntPtr handle)
    { _handle = handle; }

    ~OSHandle()
    { CloseHandle(_handle); }

    [System.Runtime.InteropServices.DllImport("Kernel32")]
    private extern static bool CloseHandle(IntPtr handle);
}
```



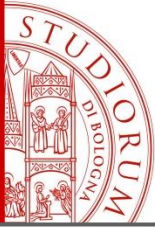
Finalization

- Il compilatore C# trasforma il codice del distruttore

```
~OSHandle()  
{ CloseHandle(_handle); }
```

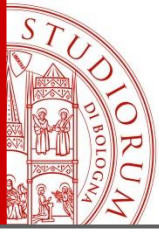
nel seguente codice (ovviamente in IL):

```
protected override void Finalize()  
{  
    try  
    { CloseHandle(_handle); }  
    finally  
    { base.Finalize(); }  
}
```



Finalization

- L'invocazione del metodo **Finalize** non avviene in modo deterministico
- Inoltre, non essendo un metodo pubblico, il metodo **Finalize** non può essere invocato direttamente
- Nel caso di utilizzo di risorse che devono essere rilasciate appena termina il loro uso, questa **situazione è problematica**
- Si pensi a file aperti o a connessioni a database che vengono chiusi solo quando il GC invoca il corrispondente metodo **Finalize**
- In questi casi, è di fondamentale importanza rilasciare (**Dispose**) o chiudere (**Close**) la risorsa in **modo deterministico**



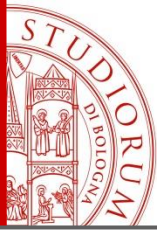
Rilascio deterministico

senza gestione eccezioni 😊

...

```
Byte[] bytesToWrite = new Byte[] {1,2,3,4,5};  
FileStream fs;  
fs = new FileStream("Temp.dat", FileMode.Create);  
fs.Write(bytesToWrite, 0, bytesToWrite.Length);  
fs.Close();
```

...



Rilascio deterministico

con gestione eccezioni 😊

```
...  
Byte[] bytesToWrite = new Byte[] {1,2,3,4,5};  
FileStream fs = null;  
try  
{  
    fs = new FileStream("Temp.dat", FileMode.Create);  
    fs.Write(bytesToWrite, 0, bytesToWrite.Length);  
}  
finally  
{  
    if(fs != null) fs.Close();  
}  
...
```



Il pattern Dispose

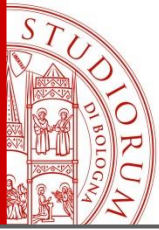
- Se un tipo **T** vuole offrire ai suoi utilizzatori un servizio di ***clean up* esplicito**, deve implementare l'interfaccia **IDisposable**

```
public interface IDisposable
{
    void Dispose();
}
```

- I clienti di **T** possono utilizzare l'istruzione **using**

```
using (T tx = ...)
{
    utilizzo di tx...
}
```

Invocazione automatica di **tx.Dispose()**



Rilascio deterministico

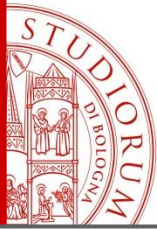
con using 😊😊

...

```
Byte[] bytesToWrite = new Byte[] {1,2,3,4,5};  
using (FileStream fs =  
    new FileStream("Temp.dat", FileMode.Create))  
{  
    fs.Write(bytesToWrite, 0, bytesToWrite.Length);  
}
```

...

- Il tipo della variabile definita nella parte iniziale di **using** deve implementare l'interfaccia **IDisposable**
- All'uscita del blocco **using**, viene sempre invocato automaticamente il metodo **Dispose**



Il pattern Dispose

altro esempio di utilizzo

```
public class CursorReplacer : IDisposable
{
    private readonly Cursor _previous;

    public CursorReplacer()
    {
        _previous = Cursor.Current;
        Cursor.Current = Cursors.WaitCursor;
    }

    public void Dispose()
    {
        Cursor.Current = _previous;
    }
}
```




Il pattern Dispose

altro esempio di utilizzo

```
List<DbTableWrapper> tableWrappers = new List<DbTableWrapper>();  
// Recupero di tutte le tabelle selezionate  
using (CursorReplacer cursorReplacer = new CursorReplacer())  
{  
    foreach (DbServerWrapper serverWrapper in  
        SelectedDbServerWrappers)  
        foreach (DbCatalogWrapper catalogWrapper in  
            serverWrapper.SelectedDbCatalogWrappers)  
            foreach (DbTableWrapper tableWrapper in  
                catalogWrapper.SelectedDbTableWrappers)  
            {  
                tableWrappers.Add(tableWrapper);  
            }  
}
```